Unification with Flexible Arity Symbols: a Typed Approach

Jorge Coelho[1] and Mário Florido[2]

[1] Instituto Superior de Engenharia do Porto & LIACC
Porto, Portugal
[2] University of Porto, DCC-FC & LIACC
Porto, Portugal
{jcoelho,amf}@ncc.up.pt

Abstract. In this paper we extend unification of terms with flexible arity function symbols with regular expression operators such as repetition (*), alternation, etc, that are used as types allowing a compact representation of terms with functors with an arbitrary number of arguments. This new unification can then be used to unify subterms in the middle of complexes sequences of terms leading to a quite simple representation of sequences of arguments.

1 Introduction

In this paper we present a new form of unification, here called *regular expression unification*, based on unification of terms with flexible arity function symbols [15]. Its extra expressiveness comes from the use of regular expressions types to dynamically unify terms. Let us present an illustrating example.

The following declaration introduces regular expression types describing terms in a simple bibliographic database:

$$\begin{array}{ll} \alpha_{bib} & \longrightarrow \{bib(\alpha_{book}+)\} \\ \alpha_{book} & \longrightarrow \{book(\alpha_{author}+, \alpha_{name})\} \\ \alpha_{author} & \longrightarrow \{author(string)\} \\ \alpha_{name} & \longrightarrow \{name(string)\} \end{array}$$

Type expressions of the form $f(...)$ classify tree nodes with the label f. Type expressions of the form $t*$ denote a sequence of arbitrary many ts, and $t+$ denote a sequence of arbitrary many ts with at least one t. Thus terms with type α_{bib} have *bib* as functor and their arguments are a sequences of one or more books. Terms with type α_{book} have *book* as functor and their arguments are a sequence consisting of one or more authors followed by the name of the book.

The next type describes arbitrary sequences of authors with at least two authors:

$$\alpha_a \longrightarrow \{(author(string), author(string), author(string)*)\}$$

To get the names of all the books with two or more authors we just need the following unification ($= * =$ stands for our regular expression unification and $t :: \tau$ means that term t has type τ):

$$bib(\_,book(X::\alpha_a,name(N)),\_) =^*= BibDoc::\alpha_{bib}.$$

This unifies two terms typed by α_{bib}. The type declaration in the first argument is not needed because it can be easily reconstructed from the term. In this case we bind the variable N to the name's content of the first book with at least two authors. Note how the type α_a in the first argument of the unification is used to jump over an arbitrary number of arguments and extract the name of the first book with at least two authors. All the results can then be obtained, one by one, by backtracking. This style of unification goes far beyond standard term unification.

The main contributions of this work are:

1. *Sequence types:* sequence types are types of sequences of arbitrary length and play a fundamental role in our work because they are the internal representation of *regular expression types*. We define sequence types and then define a new algorithm to calculate the intersection of any two given sequence types.
2. the definition of a *unification* algorithm for terms with functors of flexible arity typed by regular expression types.

In this paper the theorems are presented without proofs. Proofs of every theorem can be found in an extended version [6]. We start in section 2 by presenting the related work, then in section 3 we introduce sequence variables and flexible arity functions. In section 4 we introduce types and the intersection algorithm, and in section 5 we present Regular Expression Unification. Finally in section 6 we present some simple examples of the application of Regular Expression Unification to XML processing.

2 Related Work

Languages with flexible arity symbols have been used in various areas: Kutsia ([15]) defined a procedure for unification with sequence variables and flexible arity symbols applied to an extension of *Mathematica* for mathematical proving ([4]). In [5] unification of terms with flexible arity functor was used as the base of the constraint logic programming language CLP(Flex). The Knowledge Interchange Format KIF ([12]) and the tool Ontolingua [10] extend first order logic with variable arity function symbols and apply it to knowledge management. Feature terms [21] can also be used to denote terms with flexible arity and have been used in logic programming, unification grammars and knowledge representation. Unification for flexible terms has as particular instances previous work on word unification ([14, 20]), equations over lists of atoms with a concatenation operator ([8]) and the work of Makanin for solving equations over free semigroups ([17]). In all the works cited before unification is untyped. This work presents a typed version of this kind of unification, where types, besides there

usual use as the guarantee of correctness accordingly to some pre-defined type information, give a quite compact specification of sequences of arguments with a similar structure.

The system presented in [23] used types together with a novel form of unification (simulation unification [3]) for a small subset of the logic XML query language Xcerpt [2]. The previous work extended tree automata to deal directly with terms with functors of arbitrary arity. In our work we do not need such extension because the arbitrary number of arguments in our framework is denoted by the notion of *sequence* that is expressible using two symbols of fixed arity 2 and 0. Some previous work (including [23]) uses tree automata for typing XML query languages. Tree automata recognize languages defined by regular tree grammars ([22]) that correspond to regular types [9], a framework widely used as type language for logic programming [25, 11, 9]. In this paper we follow the regular type approach.

Functional languages for XML processing (such as XDuce ([13]) and CDuce ([1]) relied on the notion of trees with an arbitrary number of leaf nodes. However they dealt with pattern matching, not unification. Regular expression patterns are often ambiguous and thus presume a fixed matching strategy for making the matching deterministic. In contrast, our work just leaves ambiguous matching non-deterministic and examines every possible matching case by backtracking.

Regular expression matching was also used in [16] to extend context sequence matching with context and sequence variables. This previous work dealt with matching, not unification, and it was not integrated in a programming language.

There are other approaches to the unification of infinite terms based on rational trees [7], but in these approaches the problem addressed is different from ours; rational trees deal with trees with an infinite depth and our approach deals with trees where each node may have an infinite number of children.

3 Sequence Variables and Flexible Arity Functions

Here we introduce the notion of finite sequence of term as it was done before in [15, 5].

Definition 1. *A* sequence $\tilde{t}$*, is defined as follows:*

- ϵ *is the empty sequence.*
- $t_1, \tilde{t}$ *is a sequence if* t_1 *is a term and* $\tilde{t}$ *is a sequence*

Example 1. Given the terms $f(a)$, b and X, then $\tilde{t} = f(a), b, X$ is a sequence.

Consider an alphabet consisting of the following sets: the set of standard variables, the set of sequence variables, the set of constants, the set of fixed arity function symbols and the set of flexible arity function symbols.

Definition 2. *The set of terms over the previous alphabet is the smallest set that satisfies the following conditions:*

1. *Constants and sequence variables are terms.*
2. *If f is a flexible arity function symbol and $t_1, \ldots, t_n$ ($n \geq 0$) are terms, then $f(t_1, \ldots, t_n)$ is a term.*

We use a special kind of terms, here called *sequence terms*, for implementing sequences.

Definition 3. *A* sequence term*, $\bar{t}$ is defined as follows:*

- ϵ *is a* sequence term *that represents the empty sequence.*
- $seq(t, \bar{s})$ *is a* sequence term *if t is a term and $\bar{s}$ is a* sequence term*.*

Definition 4. *A sequence term in* normal form *is defined as:*

- ϵ *is in normal form*
- $seq(t_1, t_2)$ *is in normal form if t_1 is not of the form $seq(t_3, t_4)$ and t_2 is in normal form.*

Note that sequence terms in normal form are our internal notation for sequences. For example, $seq(a, seq(b, \epsilon))$ is the same as a, b.

Definition 5. *Given the sequence terms $\bar{t}_1$ and $\bar{t}_2$, we define sequence term* concatenation *as $\bar{t}_1 + +\bar{t}_2$, where the $++$ operator is defined as follows:*

$$\begin{array}{rl} \epsilon & ++\ \bar{t} = \bar{t} \\ seq(t_1,\bar{t_2}) & ++\ \bar{t_3} = seq(t_1,\bar{t_2}++\bar{t_3}) \end{array}$$

Definition 6. *Given a sequence, we define sequence* normalization *as:*

$$\begin{array}{ll} norm(\epsilon) & = \epsilon \\ norm(f(\bar{t})) & = seq(f(norm(\bar{t})),\epsilon),\ \text{if } \bar{t} \text{ is a sequence term.} \\ norm(t) & = seq(t,\epsilon),\ \text{if } t \text{ is a term not of the form } seq(t_1,\bar{t}). \\ norm(seq(t_1,\bar{t})) & = norm(t_1)\ ++\ norm(\bar{t}) \end{array}$$

In our setting, a term $f(t_1, t_2, \ldots, t_n)$, where f has flexible arity, is internally represented as $f(seq(t_1,\ seq(t_2, \ldots,\ seq(t_n, \epsilon),\ \ldots))$, that is, arguments of functions of flexible arity are always represented as elements of a sequence term in normal form. Also note that the previous functions and definitions are all polymorphic in the sense that they apply to sequences of objects of any type. In fact in this paper we will use *seq*, *norm* and $++$ applied to types (in section 4.2) and to terms (in section 5).

4 Types

4.1 Regular Types

The next definitions and examples introduce briefly the notion of Regular Types along the lines presented in [9].

Definition 7. *Assuming an infinite set of* type symbols*, a* type term *is defined as follows:*

1. *A constant symbol is a type term* $(a, b, c, ...)$.
2. *A type symbol is a type term* $(\alpha, \beta, ...)$
3. *If* f *is a flexible arity function symbol and each* τ_i *is a type term,* $f(\tau_1, ..., \tau_n)$ *is a type term.*

Definition 8. *A* type rule *is an expression of the form* $\alpha \rightarrow \Upsilon$ *where* α *is a type symbol and* Υ *is a finite set of type terms.*

Example 2. Let α and β be type symbols, $\alpha \rightarrow \{a, b\}$ and $\beta \rightarrow \{nil, tree(\beta, \alpha, \beta)\}$ are type rules.

Definition 9. *A type symbol* α *is defined by a set of type rules* T *if there exists a type rule* $\alpha \rightarrow \Upsilon \in T$.

We make some assumptions:

1. The constant symbols are partitioned in non-empty subsets, called *base types*. Some examples are, *string*, *int*, and *number*.
2. The existence of μ, the universal type, and ϕ representing the empty type.
3. Each type symbol occurring in a set of type rules T is either μ, ϕ, a base type symbol, or a type symbol defined in T, and each type symbol defined in T has exactly one defining rule in T.

Regular types are the class of types that can be specified by finite sets of type rules.

4.2 Sequence types

We now use the functor *seq* defined in section 3 to extend the previous type language to type sequences of terms: A *sequence type* is ϵ or $seq(\tau, \bar{\tau})$ where τ is a type and $\bar{\tau}$ is a sequence type.

Example 3. Given the type symbol α_{seq}, functor f, constant c and type rule $\alpha_{seq} \rightarrow \{\epsilon, seq(f(seq(c, \epsilon)), \alpha_{seq})\}$, α_{seq} describes sequences of elements of the form $seq(f(seq(c, \epsilon)), \epsilon)$, such as ϵ, $seq(f(seq(c, \epsilon)), \epsilon)$, $seq(f(seq(c, \epsilon)), seq(f(seq(c, \epsilon)), \epsilon))$, ...

We add sequence types to our type language. Thus, from now on, when we refer to a type we mean a regular type or a sequence type.

4.3 Regular Expression Types

Regular Expression Types describe sequences of values. We have a notation for several kinds of sequences of values (*,+,?,| and ,). Note that a sequence of values is also a value described by regular expression types. Given types a and b, the following table describes *regular expression types*:

$a*$ sequence of zero or more a's
$a+$ sequence of one or more a's
$a?$ zero or one a
$a|b$ a or b
a,b a followed by b.

We translate regular expression types to our internal sequence notation. The translation is made accordingly to the rules presented ahead:

$$
\begin{array}{l}
a* \Rightarrow \alpha_* \rightarrow \{\epsilon, seq(a, \alpha_*)\} \\
a+ \Rightarrow \alpha_+ \rightarrow \{a, seq(a, \alpha_+)\} \\
a? \Rightarrow \alpha_? \rightarrow \{\epsilon, a\} \\
a|b \Rightarrow \alpha_| \rightarrow \{a, b\} \\
a, b \Rightarrow \alpha_{seq} \rightarrow \{seq(a, seq(b, \epsilon))\}
\end{array}
$$

Definition 10. *A* type declaration *for a term t with respect to a set of type rules T is a pair $t :: \alpha$ where α is a type symbol defined in T.*

Note that in any term t, it is only necessary to declare types for variables since this is enough to reconstruct the type for t. When a term t is typed by μ (the universal type) we will omit the type declaration. Thus, in the rest of the paper when a term appear without its type declaration it is assumed that it is typed by the universal type μ. We now define the notion of typed unification.

Definition 11. *If t_1 and t_2 are terms and α_1 and α_2 are types, then $t_1 :: \alpha_1 = * = t_2 :: \alpha_2$ denotes unification of typed terms with flexible arity symbols.*

Terms are interpreted on a semantic domain of trees over uninterpreted functors. Equality between trees is defined in the standard way: two trees are equal if and only if their root functor are the same and their corresponding subtrees, if any, are equal. An equation $t_1 :: \alpha_1 = * = t_2 :: \alpha_2$ is solvable if and only if there is an assignment of sequences or ground terms, respectively, to variables therein such that the the terms become equal and the corresponding types become equal and are not empty.

Example 4. Consider the equation $a(X, b, Y) :: \alpha_a = * = a(a, b, b, b) :: \mu$, where α_a is defined by the type rules:

$$
\begin{array}{l}
\alpha_a \longrightarrow \{a(\mu, b, \alpha_y)\} \\
\alpha_y \longrightarrow \{b, (b, \alpha_y)\}
\end{array}
$$

then the unification gives two results:

1. $X = a$ and $Y = b, b$
2. $X = a, b$ and $Y = b$

Note that without the types the solution $X = a, b, b$ and $Y = \epsilon$ would also be valid.

4.4 Type intersection

The intersection of types plays an important role in our new unification algorithm. In this section we present some definitions, the intersection algorithm, correctness results and examples.

Empty types are types that only describe the empty set. A useful function is *empty*, which returns $true$ if a type is empty and $false$ otherwise.

The function *empty* and its correctness is presented in [26].

We are now able to present the type intersection algorithm. Note that the algorithm uses function *norm* presented in section 3. We use two global variables, T with the initial set of type rules and I which stores the intersections already made in order to avoid cycles. Whenever more than one case is applicable the first one is used. The intersection algorithm is described in figure 1

Example 5. Let us calculate the intersection of $seq(\alpha, \epsilon) \cap seq(a, \alpha)$ with $T = \{\alpha \rightarrow \{\epsilon, seq(a, \alpha\} \}$ and $I = \{\}$. This corresponds to the case number 9, $seq(\alpha, \epsilon) \cap seq(a, \alpha) = \alpha_f$, where $I = \{ (seq(\alpha, \epsilon), seq(a, \alpha) , \alpha_f) \}$. Two cases follow:

1. $norm(seq(\epsilon, \epsilon)) \cap norm(seq(a, \alpha)) = \alpha_{f1}$. This results in $\epsilon \cap seq(a, \alpha) = \phi$, thus $\alpha_{f1} = \phi$.
2. $norm(seq(seq(a, \alpha), \epsilon)) \cap norm(seq(a, \alpha)) = \alpha_{f2}$. Then by case 1, $seq(a, seq(\alpha, \epsilon)) \cap seq(a, seq(\alpha, \epsilon)) = seq(a, seq(\alpha, \epsilon))$, thus $\alpha_{f2} = seq(a, seq(\alpha, \epsilon))$

Thus the result is $\alpha_f \rightarrow \{seq(a, seq(\alpha, \epsilon))\}$, with $T = \{\alpha \rightarrow \{\epsilon, seq(a, \alpha\} \}$.

Definition 12. *Given a type α defined by the set of rules T, let $[\alpha]_T$ be the set of terms with type α.*

Theorem 1. *Let α_1 and α_2 be type terms defined by a set of type rules T. Then $\alpha_1 \cap \alpha_2$ terminates and returns α_f and T' and $[\alpha_f]_{T'} = [\alpha_1]_T \cap [\alpha_2]_T$.*

5 Unification

Regular Expression Unification extends unification of terms with functors of arbitrary arity [15, 5] by allowing unification only when the intersection of declared types is not empty. Kutsia showed in [15] that the untyped unification of the algorithm terminated if it had a cycle check, (i.e. it stopped with failure if a unification problem gave rise to a similar unification problem) and if each sequence variable does not occur more than twice in a given unification problem. We also have the same restriction in the number of occurrences of a variable but we don't need to implement the cycle check since we use backtracking.

This algorithm consists of two main steps, *Projection* and *Transformation*. The output of *Projection* are the inputs of *Transformation*. The first step, *Projection* is where some variables are erased from the sequence. This is needed to obtain solutions where those variables are instantiated by the empty sequence.

[1] == denotes syntactic equality

(1) $X \cap Y = X$, if $X == Y$ [1]

(2) $X \cap Y = Z$, if $(X, Y, Z) \in I$ or $(Y, X, Z) \in I$

(3) $X \cap seq(\mu, \epsilon) = X$

(4) $seq(\mu, \epsilon) \cap Y = Y$

(5) $f(\tau_1) \cap f(\tau_2) = f(\tau_1 \cap \tau_2)$

(6) $seq(\tau_1, \bar{\tau_1}) \cap seq(\tau_2, \bar{\tau_2}) = norm(seq(\tau_1 \cap \tau_2, \bar{\tau_1} \cap \bar{\tau_2}))$

(7) $seq(\alpha, \epsilon) \cap \epsilon = \epsilon$,
if
$(seq(\alpha, \epsilon), \epsilon, \alpha') \notin I$,
$I = I \cup \{(seq(\alpha, \epsilon), \epsilon, \epsilon)\}$,
$\alpha \rightarrow \{\alpha_1, \alpha_2, \ldots, \alpha_n\} \in T$
and $\exists_i . norm(\alpha_i) \cap \epsilon = \epsilon$

(8) $\epsilon \cap seq(\alpha, \epsilon) = \epsilon$,
if
$(seq(\alpha, \epsilon), \epsilon, \alpha') \notin I$,
$I = I \cup \{(seq(\alpha, \epsilon), \epsilon, \epsilon)\}$,
$\alpha \rightarrow \{\alpha_1, \alpha_2, \ldots, \alpha_n\} \in T$
and $\exists_i . norm(\alpha_i) \cap \epsilon = \epsilon$

(9) $seq(\alpha, \bar{\tau_1}) \cap seq(\tau_2, \bar{\tau_2}) = \alpha_f$, where α_f is a new type symbol and given that
$\alpha \rightarrow \{\alpha_1, \alpha_2, \ldots, \alpha_n\} \in T$,
$I = I \cup \{(seq(\alpha, \bar{\tau_1}), seq(\tau_2, \bar{\tau_2}), \alpha_f)\}$, $NR = \{\}$
for each α_i,
$norm(seq(\alpha_i, \bar{\tau_1})) \cap seq(\tau_2, \bar{\tau_2}) = \alpha_i$,
if $empty(\alpha_i, \{\}) = false$
then $NR = NR \cup \{\alpha_i\}$, $1 \leq i \leq n$
end_for_each
if NR is $\{\}$ then $\alpha_f = \phi$
else $T = T \cup \{\alpha_f \rightarrow NR\}$

(10) $seq(\tau_1, \bar{\tau_1}) \cap seq(\alpha, \bar{\tau_2}) = \alpha_f$, where α_f is a new type symbol and given that
$\alpha \rightarrow \{\alpha_1, \alpha_2, \ldots, \alpha_n\} \in T$,
$I = I \cup \{(seq(\tau_1, \bar{\tau_1}), seq(\alpha, \bar{\tau_2}), \alpha_f)\}$, $NR = \{\}$
for each α_i,
$norm(seq(\alpha_i, \bar{\tau_1})) \cap seq(\tau_1, \bar{\tau_2}) = \alpha_{fi}$,
if $empty(\alpha_i, \{\}) = false$
then $NR = NR \cup \{\alpha_i\}$, $1 \leq i \leq n$
end_for_each
if NR is $\{\}$ then $\alpha_f = \phi$
else $T = T \cup \{\alpha_f \rightarrow NR\}$

(11) $seq(\alpha, \bar{\tau_1}) \cap seq(\beta, \bar{\tau_2}) = \gamma$, where γ is a new type symbol and given that,
$\alpha \rightarrow \{\alpha_1, \alpha_2, \ldots, \alpha_n\} \in T$,
$\beta \rightarrow \{\beta_1, \beta_2, \ldots, \beta_m\} \in T$,
$I = I \cup \{(seq(\alpha, \bar{\tau_1}), seq(\beta, \bar{\tau_2}), \gamma)\}$, $NR = \{\}$
for each α_i and β_j
$norm(seq(\alpha_i, \bar{\tau_1})) \cap norm(seq(\beta_j, \bar{\tau_2})) = \gamma_k$,
if $empty(\gamma_k,) = false$,
$then\, NR = NR \cup \{\gamma_k\}$, $1 \leq k \leq w$ where $w = n * m$,
end_for_each
if NR is $\{\}$ then $\gamma = \phi$
else $T = T \cup \{\gamma \rightarrow NR\}$

(12) $\tau_1 \cap \tau_2 = \phi$, if none of the previous rules is applicable.

Fig. 1. Intersection of sequences of types

The second step, *Transformation* is defined by a set of rules where the non-standard unification is translated to standard first-order term unification. We consider that upper case letters (X,Y,...) stand for sequence variables, lower case letters (s,t,...) for terms and overlined lower case letters ($\bar{t}, \bar{s}$) for *sequence terms*.

Example 6 (projection). Let $T_1 = f(b, Y, f(X))$ and $T_2 = f(X, f(b, Y))$ and let A be the set of variables erased from the unification. In the projection step we obtain the following cases (corresponding to $A = \{\}$, $A = \{X\}$, $A = \{Y\}$ and $A = \{X, Y\}$):

- $T_1 = f(b, Y, f(X)), T_2 = f(X, f(b, Y))$
- $T_1 = f(b, Y, f), T_2 = f(f(b, Y))$
- $T_1 = f(b, f(X)), T_2 = f(X, f(b))$
- $T_1 = f(b, f), T_2 = f(f(b))$

As it was mentioned before, we assume type declarations only for variables in terms and sequences. In [15] it is shown that each unification problem generates a tree. Each branch of the tree corresponds to a finite computation leading to one of two possible results: success if the unification algorithm ends returning a substitution and failure if the unification fails. These trees may have an infinite number of branches. Each successful branch yields some answer composed by the set $\Theta = \{ \theta_1, \ldots, \theta_n\}$ where each θ_i is a substitution associated with step i of the computation described by the branch.

An unification problem succeeds if at least one branch of the associated tree is successful.

To present unification we need the following auxiliary definitions.

Definition 13. *Given a sequence term (that may have type annotations), the* type *function returns its type.*

$$
\begin{array}{ll}
type(\epsilon) & = \epsilon \\
type(seq(X :: \alpha, \bar{s})) & = seq(\alpha, type(\bar{s})),\ X \text{ is a variable and } \bar{s} \text{ is a sequence} \\
type(seq(f(\bar{t}), \bar{s})) & = seq(f(type(\bar{t})), type(\bar{s})),\ \bar{t} \text{ and } \bar{s} \text{ are sequences} \\
type(seq(c, \bar{s})) & = seq(c, type(\bar{s})),\ c \text{ is a constant and } \bar{s} \text{ is a sequence}
\end{array}
$$

For example, $type(seq(X :: \alpha, \epsilon)) = seq(\alpha, \epsilon)$

The tr function when applied to a sequence term t and a type τ, types t with τ.

Definition 14. *Here we define funcion tr:*

$$
\begin{array}{ll}
tr(c, c) & = c, \textit{ if } c \textit{ is a constant} \\
tr(c, seq(c, \epsilon)) & = c, \textit{ if } c \textit{ is a constant} \\
tr(X :: \alpha, \beta) & = X :: \beta, \textit{ } \alpha \textit{ and } \beta \textit{ are type symbols} \\
tr(X :: \alpha, \bar{t}) & = X :: \alpha_1, \textit{ where } \alpha \textit{ is a type symbol, } \bar{t} \textit{ is a sequence term or} \\
 & \quad \textit{constant and } \alpha_1 \textit{ is a new type symbol defined by } \alpha_1 \rightarrow \{\bar{t}\} \\
tr(f(\bar{t}_1), f(\bar{t}_2)) & = f(tr(\bar{t}_1, \bar{t}_2)) \\
tr(seq(t, \epsilon), s) & = seq(tr(t, s), \epsilon), \textit{ if } s \textit{ is not of the form } seq(s_1, \bar{s}) \\
tr(\bar{t}, \alpha) & = \bar{t}' \textit{ where } \alpha \rightarrow \{\alpha_1, \ldots, \alpha_n\} \textit{ and} \\
 & \quad tr(\bar{t}, \alpha_1) = \bar{t}'_1 \\
 & \quad \vdots \\
 & \quad tr(\bar{t}, \alpha_n) = \bar{t}'_n \\
 & \quad \textit{where} \\
 & \quad vars(\bar{t}'_1) = \{X_1 :: \beta_{11}, \ldots, X_n :: \beta_{1n}\} \\
 & \quad \vdots \\
 & \quad vars(\bar{t}'_n) = \{X_1 :: \beta_{1n}, \ldots, X_n :: \beta_{nn}\} \\
 & \quad \textit{for each } X_i \textit{ create a new type symbol } \gamma_i \textit{ where} \\
 & \qquad \gamma_i \rightarrow \{\beta_{i1}, \ldots, \beta_{in}\}, \textit{ now } \bar{t}' \textit{ is } \bar{t}, \textit{ where each type associated} \\
 & \qquad \textit{with each variable } X_i \textit{ is replaced by the corresponding } \gamma_i \\
tr(seq(t_1, \bar{t}), seq(s_1, \bar{s})) & = seq(tr(t_1, s_1), tr(\bar{t}, \bar{s}))
\end{array}
$$

Example 7. Given a sequence of *a's* or *b's* whith an a at the head represented by $seq(a, seq(X :: \alpha))$ where $\alpha \rightarrow \{\epsilon, seq(a, \epsilon), seq(b, \epsilon)\}$, its type can be restricted to a sequence of one or more *a's* given by $seq(a, seq(\alpha_1, \epsilon))$ where, $\alpha_1 \rightarrow \{\epsilon, seq(a, \epsilon)\}$. This is done by applying tr:

$$
\begin{array}{ll}
tr(seq(a, seq(X :: \alpha, \epsilon)), seq(a, seq(\alpha_1, \epsilon))) = \\
seq(tr(a, a), tr(seq(X :: \alpha, \epsilon), seq(\alpha_1, \epsilon))) & = \\
seq(a, seq(tr(X :: \alpha, \alpha_1), tr(\epsilon, \epsilon))) & = \\
seq(a, seq(X :: \alpha_1, \epsilon))
\end{array}
$$

as a result X is now typed by α_1.

Theorem 2. *Given a sequence $\bar{t}$ and a type α, which results from the intersection of type($\bar{t}$) with another sequence type, then:*

$$tr(\bar{t}, \alpha) = \bar{s} \Rightarrow type(\bar{s}) = \alpha$$

We now present the transformation algorithm. Note that transformation uses the *norm* function to normalize sequences of terms. The transformation algorithm is presented as a set of rewrite rules in figure 2. When none of the rules is applicable the algorithm fails. Note that $=$ stands for standard Robinson unification [18] and that rules 6, 7, 8 and 9 are non-deterministic: for example rule 6 states that in order to solve $seq(X :: \alpha, \bar{t}) = * = seq(s_1, \bar{s})$ we can solve $\bar{t} = * = \bar{s}$ with $X = tr(norm(s_1), \alpha_1)$ or we can solve $norm(seq(X_1 :: \beta_1, \bar{t})) = * = norm(\bar{s})$ with $X = tr(seq(s_1, seq(X_1 :: \beta, \epsilon)), \alpha_1)$. At the end the solutions given by the algorithm are normalized by the normalization function.

Example 8. Given the variable X and the type α, with type rule $\alpha \rightarrow \{\epsilon, seq(a, \alpha)\}$, let us calculate $seq(X :: \alpha, \epsilon) = * = seq(a, seq(a, \epsilon))$ By rule number 6 the algorithm first tries:

Success

(1) $t \quad =*= \quad s \quad \Longrightarrow$ True, if $t == s$

(2) $X :: \alpha \quad =*= \quad t \quad \Longrightarrow X = tr(norm(t), \alpha_1)$ if X does not occur in t,
$\text{seq}(\alpha, \epsilon) \cap type(t) = \alpha_1$,
$empty(\alpha_1) = false$

(3) $t \quad =*= \quad X :: \alpha \quad \Longrightarrow X = tr(norm(t), \alpha_1)$ if X does not occur in t,
$seq(\alpha, \epsilon) \cap type(t) = \alpha_1$,
$empty(\alpha_1) = false$

Eliminate

(4) $f(\bar{t}) \quad =*= \quad f(\bar{s}) \quad \Longrightarrow \bar{t} =*= \bar{s}$

(5) $seq(t_1, \bar{t}) \quad =*= \quad seq(s_1, \bar{s}) \quad \Longrightarrow t_1 =*= s_1$,
$\bar{t} =*= \bar{s}$

(6) $seq(X :: \alpha, \bar{t}) \quad =*= \quad seq(s_1, \bar{s}) \quad \Longrightarrow X = tr(norm(s_1), \alpha_1)$,
X does not occur in s_1,
$seq(\alpha, \epsilon) \cap type(seq(s_1, \epsilon)) = \alpha_1$,
$empty(\alpha_1) = false$,
$\bar{t} =*= \bar{s}$.
$\Longrightarrow X = tr(seq(s_1, seq(X_1 :: \beta, \epsilon)), \alpha_1)$,
where X_1 is a new variable,
β a new type symbol defined by $\beta \rightarrow \{\mu\}$,
β_1 the new type of X_1 in X,
$\alpha_1 = seq(\alpha, \epsilon) \cap type(seq(s_1, seq(X_1 :: \beta, \epsilon)))$,
$empty(\alpha_1) = false$,
$norm(seq(X_1 :: \beta_1, \bar{t})) =*= norm(\bar{s})$.

(7) $seq(t_1, \bar{t}) \quad =*= \quad seq(X :: \alpha, \bar{s}) \quad \Longrightarrow X = tr(norm(t_1), \alpha_1)$,
X does not occur in t_1,
$seq(\alpha, \epsilon) \cap type(seq(t_1, \epsilon)) = \alpha_1$,
$empty(\alpha_1) = false$,
$\bar{t} =*= \bar{s}$.
$\Longrightarrow X = tr(seq(t_1, seq(X_1 :: \beta, \epsilon)), \alpha_1)$,
where X_1 is a new variable,
β a new type symbol defined by $\beta \rightarrow \{\mu\}$,
β_1 the new type of X_1 in X,
$\alpha_1 = seq(\alpha, \epsilon) \cap type(seq(t_1, seq(X_1 :: \beta, \epsilon)))$,
$empty(\alpha_1) = false$,
$norm(\bar{t}) =*= norm(seq(X_1 :: \beta_1, \bar{s}))$,

(8) $seq(X :: \alpha, \bar{t}) \quad =*= \quad seq(Y :: \beta, \bar{s}) \quad \Longrightarrow X :: \gamma = Y :: \gamma$, where
$\gamma = seq(\alpha, \epsilon) \cap seq(\beta, \epsilon), empty(\gamma) = false$,
$\bar{t} =*= \bar{s}$.
$\Longrightarrow X = tr(seq(Y :: \beta, seq(X_1 :: \beta_1, \epsilon)), \gamma)$,
where X_1 is a new variable,
β_1 is a new type symbol defined by $\beta_1 \rightarrow \{\mu\}$,
$\gamma = seq(\alpha, \epsilon) \cap type(seq(Y :: \beta, seq(X_1 :: \beta_1, \epsilon))$,
$empty(\gamma) = false$,
β_2 is the new type of X_1 in X,
$norm(seq(X_1 :: \beta_2, \bar{t})) =*= norm(\bar{s})$.
$\Longrightarrow Y = tr(seq(X :: \alpha, seq(Y_1 :: \alpha_1, \epsilon)), \gamma)$,
where X_1 is a new variable,
α_1 is a new type symbol defined by $\alpha_1 \rightarrow \{\mu\}$,
$\gamma = seq(\alpha, \epsilon) \cap type(seq(X :: \alpha, seq(Y_1 :: \alpha_1, \epsilon)))$,
$empty(\gamma) = false$,
α_2 is the new type of Y_1 in Y,
$norm(\bar{t}) =*= norm(seq(Y_1 :: \alpha_2, \bar{s}))$.

Split

(9) $seq(t_1, \bar{t}) \quad =*= \quad seq(s_1, \bar{s}) \quad \Longrightarrow$ if $t_1 =*= s_1 \Longrightarrow r_1 =*= q_1$ then
$norm(seq(r_1, \bar{t})) =*= norm(seq(q_1, \bar{s}))$
$\vdots$
$\Longrightarrow$ if $t_1 =*= s_1 \Longrightarrow r_w =*= q_w$ then
$norm(seq(r_w, \bar{t}_n)) =*= norm(seq(q_w, \bar{s}))$,
where t_1 and s_1 are compound terms.

Fig. 2. Transformation rules

- $seq(\alpha, \epsilon) \cap seq(a, \epsilon)$ returns a new type α_{f1} where $\alpha_{f1} \rightarrow \{seq(a, \epsilon)\}$. α_{f1} is not empty, thus the algorithm proceeds evaluating $\epsilon = * = seq(a, \epsilon)$ which will fail.
- Then (by the second alternative of rule 6), a new variable X_1 is created and a new type symbol β is associated to it such that $\beta \rightarrow \{\mu\}$. By the intersection algorithm, a new type symbol α_{f2} is created with type rule $\alpha_{f2} \rightarrow \{seq(a, \alpha_{f3})\}$ and $\alpha_{f3} \rightarrow \{\epsilon, seq(a, \alpha_{f3})\}$. Now $X = tr(seq(a,\ seq(X_1 :: \beta,\ \epsilon)), \alpha_{f2})$, where:

$$\begin{array}{ll} tr(seq(a, seq(X_1 :: \beta, \epsilon)), \alpha_{f2}) & = \\ tr(seq(a, seq(X_1 :: \beta, \epsilon)), seq(a, \alpha_{f3})) & = \\ seq(tr(a, a), tr(seq(X_1 :: \beta, \epsilon), \alpha_{f3})) & = \\ seq(a, seq(tr(X_1 :: \beta, \alpha_{f3}), \epsilon)) & = \\ seq(a, seq(X_1 :: \alpha_{f3}, \epsilon)) & \end{array}$$

  Now, the algorithm proceeds with the unification, $seq(X_1 :: \alpha_{f3},\ \epsilon) = * = seq(a,\ \epsilon)$. Thus it intersects $seq(\alpha_{f3},\ \epsilon)$ with $seq(a, \epsilon)$ resulting in $\alpha_{f4} \rightarrow \{seq(a, \epsilon)\}$ and X_1 is instantiated with $seq(a, \epsilon)$. The algorithm succeeds with $X = seq(a,\ seq(a, \epsilon))$.

In the following theorem we will use the notion of substitution presented in [15].

Theorem 3 (Soundness). *Let Θ be the answer substitution of the equation $t_1 = * = t_2$. Then:*

1. $\Theta t_1 = \Theta t_2$
2. $type(t_1) \cap type(t_2) = type(t)$, *where* $t = \Theta t_1 = \Theta t_2$

6 Application to XML processing

Ideas presented in this paper are used in the XML processing language XCentric (http://www.ncc.up.pt/xcentric/). Regular expression types are a model for XML grammars such as DTDs, and regular expression unification can be used in a highly declarative setting for XML processing. We now present simple application examples.

In XCentric, programs have a syntax similar to Prolog extended with the new constraint $= * =$, and type rules $\alpha \rightarrow \{\tau_1,\ \ldots,\ \tau_n\}$ are represented by the declaration :-type α - - -> $\tau_1; \ldots; \tau_n$. The operational model is the same of Prolog and we shall assume that the domain of interpretation of variables is predetermined by the context where they occur. Variables occurring in an equation of the form $t_1 = * = t_2$ are interpreted in the domain of sequences of trees, otherwise they are standard Prolog variables. Therefore, each predicate symbol, functor and variable is used in a consistent way with respect to its domain of interpretation.

DTDs (Document Type Definition) [24] can be trivially translated to regular expression types and an XML document is translated to a term with flexible arity function symbol. This term has a main functor (the root tag) and zero or

more arguments. Although our actual implementation translates attributes to a list of pairs, since attributes do not play a relevant role in this work we will omit them in the examples, for the sake of simplicity.

Example 9. Consider the simple XML file:

```
<addressbook>
        <record><name>John</name>
                <address>New York</address>
                <email>john.ny@mailserver.com</email>
        </record>
        <record><name>Sofia</name>
                <address>Rio de Janeiro</address>
                <phone>123456789</phone>
                <email>sofia.brasil@mail.br</email>
        </record>
</addressbook>
```

This XML file is valid accordingly to the following DTD:

```
<!ELEMENT addressbook (record*)>
<!ELEMENT record (name,address,phone?,email)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT email (#PCDATA)>
```

The XML file corresponds to the following term:

```
addressbook(record(name('John'),
                   address('New York'),
                   email('john.ny@mailserver.com')),
            record(name('Sofia'),
                 address('Rio de Janeiro'),
                 phone('123456789'),
                 email('sofia.brasil@mail.br'))))::type_addr
```

Where *type_addr* is described by the following type rule:

```
:-type type_addr ---> addressbook(record(name(string),address(string),
                                         phone(string)?,email(string))*)
```

From now on, whenever a variable is presented without any type information it is implicitly associated with the universal type *any* (which types any term).

Through the following examples we will use the builtin predicates *xml2pro* and *pro2xml* which respectively convert XML files into terms and vice-versa.

Example 10. We use a new syntax for sequences of elements: () stands for the empty sequence and $(e_1, \ldots, e_n)$ stands for the sequence of elements e_1 to e_n. In this example we use the address book document presented before. In this address book we have sometimes records with a phone tag. We want to build a new XML document without this tag. Thus, we need to get all the records and ignore their phone tag (if they have one). This can be done by the following program (this example is similar to one presented in XDuce [13]):

```
:-type type_a ---> addressbook(record(name(string),address(string),
                                      phone(string)?,email(string))*).
:-type type_r ---> record(name(string),address(string),email(string)).

translate:-
  xml2pro('addressbook.xml',Xml),
  process(Xml,NewXml),
  pro2xml(NewXml,'addressbook2.xml').

process(A,NewA):-
  A::type_a =*= addressbook(A2),
  r_without_phone(A2,A3),
  newdoc(addressbook,A3,NewA).

r_without_phone((_,X::type_r,S2),(X,S3)):-!,
        r_without_phone(S2,S3).

r_without_phone(_,()).
```

7 Conclusion and Future Work

In this paper we define regular expression types and show there use as a compact representation of sequences of terms in a new algorithm for the unification of terms with functor of flexible arity. There is now ongoing work applying these ideas in a statically typed version of XCentric. Another direction for future work is to enrich the type language by pushing the limits of what can be used as a compact representation of sequences of terms (languages similar to some constructions in XML-Schema [19] seem to be a good candidate).

References

1. Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming*, Uppsala, Sweden, 2003.
2. F. Bry and S. Schaffert. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In *2nd Annual International Workshop Web and Databases*, volume 2593 of *LNCS*, 2002.
3. F. Bry and S. Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *International Conference on Logic Programming (ICLP)*, volume 2401 of *LNCS*, 2002.
4. B. Buchberger, C. Dupre, T. Jebelean, B. Konev, F. Kriftner, T. Kutsia, K. Nakagawa, F. Piroi, D. Vasaru, and W. Windsteiger. The Theorema System: Proving, Solving, and Computing for the Working Mathematician. Technical Report 00-38, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, 2000.
5. Jorge Coelho and Mário Florido. CLP(Flex): Constraint Logic Programming Applied to XML Processing. In *Ontologies, Databases and Applications of SEmantics (ODBASE)*, volume 3291 of *LNCS*. Springer Verlag, 2004.

6. Jorge Coelho and Mário Florido. Unification with flexible arity symbols: a typed approach. Technical report, DCC-FC, LIACC. University of Porto, (available from www.ncc.up.pt/~jcoelho/UnifTyped.pdf), 2006.
7. A. Colmerauer. *Logic Programming*, chapter Prolog and Infinite Trees. Academic Press, 1982.
8. A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.
9. P. Dart and J. Zobel. A regular type language for logic programs. In Frank Pfenning, editor, *Types in Logic Programming*. The MIT Press, 1992.
10. A. Farquhar, R. Fikes, and J. Rice. The ontolingua server: A tool for collaborative ontology construction. *International Journal of Human-Computer Studies*, 46(6):707–727, 1997.
11. M. Florido and L. Damas. Types as theories. In *Proc. of post-conference workshop on Proofs and Types, Joint International Conference and Symposium on Logic Programming*, 1992.
12. M. R. Genesereth and R. E. Fikes. Knowledge Interchange Format, Version 3.0 Reference Manual TR Logic-92-1. Technical report, Stanford University, Stanford, 1992.
13. Haruo Hosoya and Benjamin Pierce. XDuce: A typed XML processing language. In *Third International Workshop on the Web and Databases (WebDB2000)*, volume 1997 of *LNCS*, 2000.
14. J. Jaffar. Minimal and complete word unification. *Journal of the ACM*, 37(1):47–85, 1990.
15. T. Kutsia. Unification with sequence variables and flexible arity symbols and its extension with pattern-terms. In *Joint AISC'2002 - Calculemus'2002 conference*, LNAI, 2002.
16. Temur Kutsia. Context sequence matching for xml. In *Proceedings of the 1th International Workshop on Automated Specification and Verification of Web Sites*, Valencia, Spain, 2005.
17. G. S. Makanin. The problem of solvability of equations in a free semigroup. *Math. Sbornik USSR*, 103:147–236, 1977.
18. John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
19. XML Schema. http://www.w3.org/XML/Schema/, 2000.
20. Klaus U. Schulz. Word unification and transformation of generalized equations. *Journal of Automated Reasoning*, 11(2):149–184, 1993.
21. Gert Smolka. Feature constraint logics for unification grammars. *Journal of Logic Programming*, 12:51–87, 1992.
22. J.W. Thatcher. *Tree automata: An informal survey*. Prentice-Hall, 1973.
23. Artur Wilk and Wlodzimierz Drabent. On types for xml query language xcerpt. In *Principles and Practice of Semantic Web Reasoning*, volume 2901 of *LNCS*, 2003.
24. Extensible Markup Language (XML). http://www.w3.org/XML/, 2003.
25. E. Yardeni and E. Shapiro. A type system for logic programs. In *The Journal of Logic Programming*, 1990.
26. Justin Zobel. Derivation of polymorphic types for prolog programs. In *Proc. of the 1987 International Conference on Logic Programming*. MIT Press, 1987.